Benchmarking OpenFaaS and OpenLambda with domain specific applications

Vivek Pandit, Shreyas R S, Lahari G, Sheshadri K R

Indian Institute of Science, Bangalore India,
vivekpanditdv@gmail.com, iamshrey26@gmail.com, lahari.raj.2003@gmail.com, sheshadrik@iisc.ac.in,

1 Introduction

FaaS is one of the most extensively used Cloud computing service offering. The 'serverless' nature of the FaaS platform offloads the management and administration complexity of servers to the platform provider thus making the server management transparent to the users. This along with 'pay-per-use' (billed only per function execution and no idle costs) billing model are some of the powerful features which has attracted applications from various domains. Users can now focus on designing and developing their applications rather than performing the arduous task of systems administration. The design paradigm of the platform expects the applications to be constructed in microservices architecture. The platform offers event-driven, short-lived, stateless executions of these microservices as functions. Though the FaaS platform has witnessed high adaption from the users, there is a lack of clarity on the workloads and applications that are run on these platforms. Many works feature baselines [10] and synthetic workloads to evaluate platforms, it may not reflect the exact resource behavior of the applications and workflows based on real-world usecases. It is imperative to construct real-world usecases and extract the application's characteristics *from their domain* to aid further improvements, innovations, and refinement in the FaaS platform. In this work, use-cases and applications workflows, their workloads, input characteristics are distilled from three different application domains. Analysis and profiling of functions in these workloads give us ideas and decisions on resource management, design directions, function placement in FaaS platforms

2 Application domains

In this section, three contrasting domains that have adapted FaaS are chosen to derive the baseline application scenario. The application characteristics, nature of the workload, fitness to FaaS are discussed in detail.

2.1 Video Surveillance

The video surveillance scenario is one of the popular usecases for the FaaS computing service. In this scenario, the video stream from camera sensors is processed based on the business objectives. A typical video surveillance scenario is

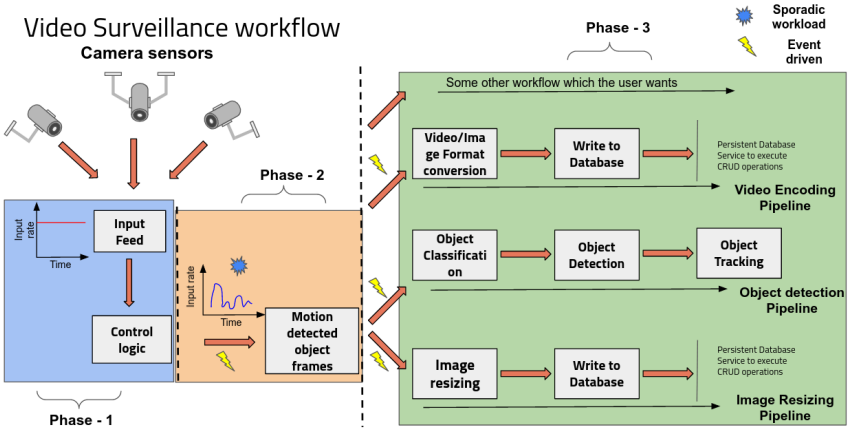


Fig. 1. Video Surveillance Workflow

shown in Fig. 1, the application can be split into three phases. The first phase has the data source, video frames from multiple camera sensors act as video source; the motion-detected frames are selected to be passed to the next phase. The second phase contains the business logic or policy, which identifies the execution pipeline. Based on the motion-detected frames, the policy mechanism triggers relevant subsequent functions. The third phase features the application pipelines, various function workflows can be added by the user to suit the surveillance needs. In the considered case three workflows are present they are Video Encoding pipeline, Object Detection pipeline, and Image Resizing pipeline.

The video stream from the camera sensors form a continuous workload, it may feature the surveillance of areas such as parking lot, reception area, etc. It is not uncommon that large portions of video streams are not relevant for processing as they may not feature any 'interesting' activity. The phases where activity of interest (such as motion detection) takes place are recorded for durations of the activity [13]. Though the video stream has a continuous workload, the application pipelines experience a sporadic workload. This is because the activity of interest (motion detection or person detection) can occur sporadically or aperiodically [13].

Why FaaS? : FaaS would be a great fit in this scenario because of the sporadic workload and event-drivenness. The functions in the application pipeline would only be triggered on the events of motion detection, FaaS can be very effective compared to the conventional resource reservation models of IaaS and CaaS. Some of the services such as the logic of motion detection, collecting the input source, the business logic may need to be kept-alive as persistent services.

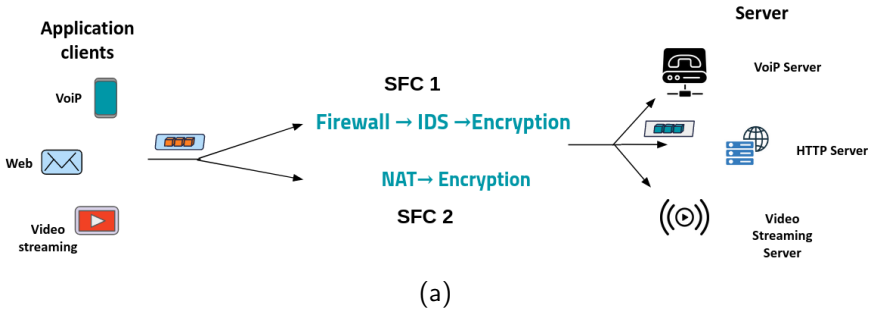


Fig. 2. Service Function Chains

2.2 Network Function Virtualization

Network functions are components of a workflow that form network services, these services are usually implemented on a specialized physical device and are tightly coupled with the hardware. Examples of network functions include Firewalls, Intrusion Detection Systems, Load Balancers etc. Network Function Virtualization (NFV) decouples the service software from the proprietary hardware and enables it to be run on virtualized computing services such as Virtual Machines (VMs), containers, or even directly as processes on commodity servers. Such network functions are called "Virtual Network Functions" (VNFs).

A typical network service is shown in Fig. 2. Source is the entity which generates packets, each packet has to pass through the functions in the Network Service. The chain of functions that form Network Service is called Service Function Chain (SFC). The Fig. 2 shows a basic Network Service which contains three Network functions Firewall(FW), Intrusion Detection System (IDS), and Encryption (Ec). Rules can be imposed on the path of the Network Service that traffic should flow through based on class of traffic [8]. In the considered example, for TCP traffic SFC2 may be taken, SFC1 is chosen for UDP traffic. The rules can be configured based on many parameters relating to trust group, multi-tenancy, packet size, along with other application characteristics as well. Common applications in this domain include Voice over Internet Protocol(VoIP), Web Services, and Video Streaming.

Why FaaS?: Previously, a lot of work has tried to address the problem of VNF placement on Virtual Machines and containers with varying optimization goals (ranging from low cost, low latency, high throughput, etc.,). These solutions often suffer from the problem of underutilizing the cloud resources when the traffic is low, similarly, they cannot scale in the presence of an aperiodic workload. In the case of Network service, some of the VNFs in the SFC exhibit workload sporadicity [2] [7]. While some need to be a persistent service as they front-line all the incoming traffic.

2.3 Web and Inferencing Applications

Inferencing is a common task in many applications in the recent times, it typically involves prediction/classification services using ML models. Examples of this involve ascertaining whether certain texts contain spam, detection of an activity, detecting an object, and so on. In the case of Web applications, inferencing is used to check whether web page contents contains abusive text, contains spam or malicious content etc. A typical scenario of the inferencing application is highlighted in Fig. 3.

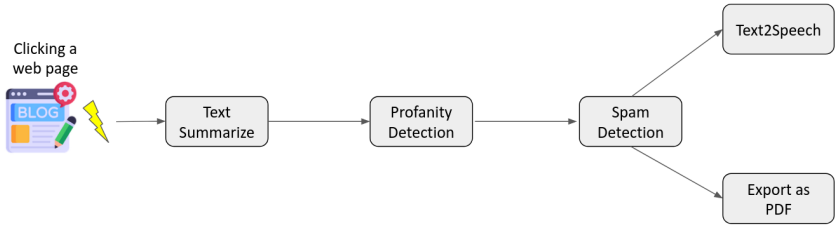


Fig. 3. Web application - Inferencing Scenario

Some of these services can also be tagged to a user facing Ecommerce application, where similar tasks described above are applied to customer product review texts.

Why FaaS? : Functions such as *SpamDetection(), ProfanityDetection()* exhbibit stateless nature, each input to them (a text document) is executed independently, and only when the webpage is requested (event-driven). Depending on the popularity of the web-page the workload can be aperiodic. Based on these features it presents a natural fit to FaaS paradigm.

3 Workload and application characteristics

The applications discussed previously all present an aperiodic workload for few of the services in the workflow. This section discusses workload distribution, the application characteristics such its resource usage, commonly accepted input etc. These findings are indented to shed light on domain-specific application characteristics and it can assist developers on deciding what part of application can benefit from FaaS.

3.1 Workload Description

Image Analytics: The video stream from camera sensors can have different resolutions, frames per second (fps), and hence can have different input sizes. Even though the video recording is continuous, the portion of interesting events

are low and aperiodic. The works [13] have reported that 0.5% to 45% of video maybe useful, most commonly 12% of the 24 hour surveillance [12] video contain useful activities. The input size has variations from 50KB to 3MB [15] [20] and even more for frames from various sensors. Apart from still motion pictures, there can also be video captures of lengths ranging from 15 seconds to 120 seconds [14]. Latency is the commonly used QoS (Quality of Service) metric for the applications in this domain.

Network: In this domain, the size of the packet and the workload intensity depends on the application. For the VoIP application, the payload size can vary from 178 bytes to 250 bytes [4] [6], with an workload intensity (packets/second) varying from 22 to 60, to even 100s. For video streaming applications the resolution of video dictates the packet size, 360p video streaming uses 3MB per minute [11] whereas 1080p video uses 15MB per minute. UDP is the protocol used for packet transmission for both UDP. The payload size will be around 1000 bytes in this case. The Web browsing application (which includes e-mail, browsing, and photo downloads) has packet sizes ranging from 50KB to 2MB [5] [11]. TCP is the preferred protocol used for packet transmission in Web Services domain. The payload size can vary from 50 bytes (for a short text) to 1000 bytes (in case of images). Jitter, latency, packet-ordering are the common QoS requirements for the applications in the NFV domain.

Inferencing applications: In this domain, the requested web page are checked for profanity, spams, and are provided other options such as text summarization, exporting as pdf. The input to these functions are text documents, the size of which can very from one web-page to another. Since web-page can also contain images, the total size can vary from 100KB to 2MB [18].

Domain	Application	Input Type	Input Size Range
Image Analytics	Object detection pipeline	Image(s)	50KB to 2MB
Network	VoIP	Packets	20 to 300
		Payload size	100 to 250
Inferencing	Text summarization	HTML web pages	50KB to 2MB, or higher

Table 1. Application Domain and Characteristics

The application domain, input characterisitics, input type, size of the input and range are specified in the Table 3.1.

4 Evaluation using Application Scenario

4.1 Representative Scenario for Image Analytics

In Image Analytics, a multi-step process orchestrates image processing through three core functions: GIF Creation, Face Detection, and Image Resizing as shown

Fig. 4. Image analytics workflow

in Fig. 4. A client triggers with image being sent by the client, the GIF Creation function undertakes the task of replicating the image thrice to produce a GIF. This resultant GIF is stored in the data store. Next, the Face Detection function is triggered through an Kafka Event Manager, it uses HarrCascades classifier to detect faces in the image. Once faces are detected, the Image Resizing function is invoked, tasked with resizing the image to a predefined specification. The data-exchange is performed through JSON, and the function output is also stored on CouchDB which is a native-JSON based data-store.

4.2 Representative Scenario for Network Service

In the NFV domain, typically network services are configured to process the network traffic going in and out of organization. As discussed in section 2.2 a network service comprises multiple network functions called Service Function Chain (SFC). We consider a representative SFC which consists of Firewall (Fw), Intrusion Detection System (IDS), and AES Encryption (Enc). The Fw performs filters packet based on source and destination IP against a 10 other IPs. The IDS compares each packet's IP against a corpus of 28 rules. Finally encryption function performs AES-256 bit encryption on the packet payload. Packets are encapsulated in HTTP requests which are sent to the FaaS controllers, as depicted in the work [19].

Fig. 5. Network Service - SFC

4.3 Representative Scenario for Web Inferencing

In the Web Inferencing domain, we present a representative workflow consisting of three functions Text Summarizer (TS), Profanity Detector (PD), and Spam-Ham Detector (SH). Text Documents are given as input to the FaaS functions by the users through HTTP requests, the TS function summarizes text content in the document, the PD function detects the presence of any abusive words; finally the SH function detects if the text contains any spam content. An Event Manager ensures the triggering of subsequent functions in the workflow.

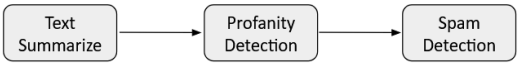


Fig. 6. Web Inferencing workflow

Subsequently, the PD function is activated through a messaging system, scanning the stored text to identify and isolate any profane language present within it. Upon detection, the SH function is engaged, responsible for distinguishing between spam and legitimate content, employing predefined criteria. The entire flow of data and interplay among these functions occurs seamlessly via serialized JSON, establishing an integrated pipeline tailored to handle and manage text within the realm of inferential operations.

5 OpenSource FaaS platforms

The preceding sections have given an overview of application domains and their workloads. In this section we evaluate a representative application from each domain on two open source FaaS platforms namely OpenFaaS and OpenLambda. The platforms are chosen based on their wide-adoption in the serverless literature, and their support for custom function runtime.

We briefly describe the architecture of OpenFaaS and OpenLambda in the following sections.

5.1 OpenLambda

OpenLambda is an open-source serverless platform designed for research prototyping. It too supports stateless function executions, supports event triggers using HTTP requests, and a limit of the execution times for a function. It supports native Python functions, and a custom runtime using Linux Containers. A detailed explanation regarding the platform is presented in Appendix C.

5.2 OpenFaaS

OpenFaaS is a popular open-source serverless platform which supports stateless execution of functions, it provides HTTP triggers as events, and a configurable timeout as an execution time-limit. It supports native program support for Python, nodeJS among many other languages, also supports a custom runtime for function execution. A detailed explanation regarding the platform, its high-level design and architecture, steps for deploying functions are stated in Appendix D.

6 Evaluation

The evaluation is performed on workflows from three different application domains namely Image Analytics, Network, Inferencing domain, and Synthetic workloads on both OpenFaaS and OpenLambda. The platforms are evaluated to understand the overheads involved in function deployment, execution times, and their support for application's domain specific requirements.

6.1 Experimental Setup

Both OpenLambda and OpenFaaS, Kafka Event Manager, CouchDB are deployed on a VM 8vCPUs and 32GB DDR4 RAM, with physical node on the Intel(R) Xeon(R) Silver 4310 CPU @ 2.10GHz node with 12 physical cores per socket with two sockets, 128 GB DDR4 RAM, and an Intel Corporation I350 Gigabit SR-IOV enabled Network port with 1Gbit/s capacity, Ubuntu 20.04 LTS was the host OS in the VM. Each function is sized to 1vCPU and 1024MB of RAM; in OpenFaaS this is 1000m CPU assignment [16].

The goal of the following experiments is to evaluate OL and OF for platform overheads, execution latency, inter function latency for the Image Analytics, Network Service, and Web Inferencing workflows.

6.2 Image Analytics

Setup : As discussed earlier in Section 3.1 the workload for the experiments reflect the variations of image sizes in the Video Surveillance Scenario. We use three image sizes (100KB, 350KB, and 750KB) as input for the Image analytics workflow to reflect the changes of images from different resolution sensors. A client from a different VM in the same network triggers HTTP requests to the Gif Creation (GC) function, to start the workflow. We discuss our observations regarding the execution times, platform overheads [1].

The plots in the Fig. 6.2 shows the end-to-end execution times for Image Analytics workflow on both OF and OL. The X-axis represents the input image sizes, the Y-axis represents the end-to-end latency in milliseconds. Both the platforms show an increased E2E latency with increased input size, the platform overheads also increase. The combined execution times also increase with increase in input size, this can be observed in the plots in Fig. 6.2. The plots in Fig. 6.2, expands on the platform overheads, it shows inter-function overheads[2] in both OF and OL for function GC $\rightarrow$ FD, and FD $\rightarrow$ IR. The platform overheads are higher in OF than in OL, this can be attributed to the movement of data through OF gateway, NATS, reads/writes to external database CouchDB, control signal to Event Manager. In OL, the data movement only happens during the invocation of function, read/writes from CouchDB data store, and control messaging to

[1] Represents additional time spent within the serverless infrastructure, database operations, and messaging; apart from executing function in the platform

[2] time between completion of one function and beginning of another

(a) End-To-End Latency (b) Function Execution Times

(c) Function Execution Times Split (d) Inter-function Latency

Fig. 7. Image Analytics Results

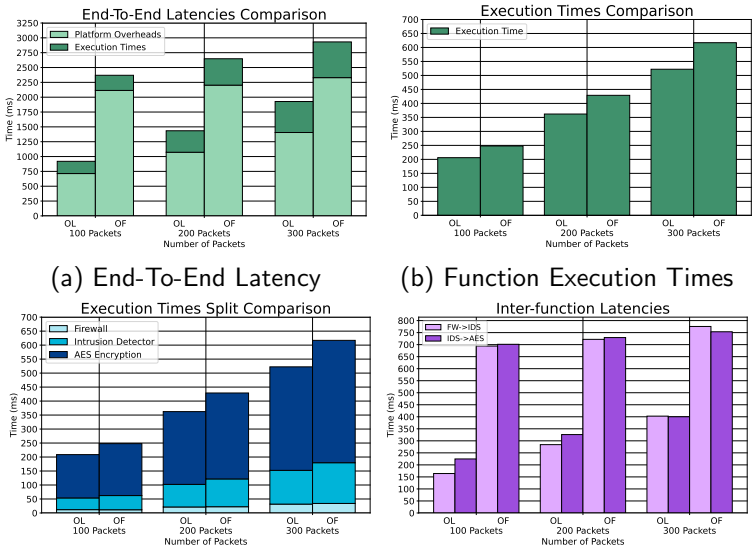
the OL controller and Event Manager. In summary, OL presents much lower overheads due to lesser data mvovement between components as compared to OF.

6.3 NFV

Setup: As discussed earlier in Section 3.1 for networking workloads, we use different flowlet or packet rate (100, 200, 300) to reflect the workloads of VoIP, and messaging applications. The payload size is set to 256 bytes. We deploy the SFC on both OF and OL. The request to trigger the SFC execution is performed by encapsulating the packets in a HTTP request payload, as suggested in [19].

The plots in the Fig. 6.3 shows the End-to-end latency for different flowlet sizes for the SFC processing. The X-axis indicates the input-size for the workflow, the Y-axis indicates the latency in milliseconds. It can be observed that the end-to-end latency increases with increase in input size (number of packets). The execution times of the SFC also increase with increase in flowlet size as observed from the plots in Fig. **??**. It can be seen that a major portion of the overheads are due to platform overheads both in OF and OL. The plots in Fig. 6.3 presents the overheads as inter-function overheads between Firewall $\rightarrow$ IDS, and IDS $\rightarrow$ Encryption. It could be seen that the overheads are significant in both OF and OL, with OF exhibiting a lot more overheads than OL. This is because of movement of data from gateway to NATS server, and final write to the CouchDB data store. The overheads increase linearly in OL with increase in flowlet, this is

contributed by the size of packet data being retrieved /stored at CouchDB data store. However, in OF the increase looks consistently higher and shows minimal changes since the movement of data dominates the overall platform overheads. Overall, both platforms show an increase in overheads with more number of packets.



(a) End-To-End Latency (b) Function Execution Times

(c) Function Execution Times Split (d) Inter-function Latency

Fig. 8. NFV and SFC Results

6.4 Web Inferencing

Setup : As discussed earlier in section 3.1 for Web workloads, we use different document sizes (10KB, 50KB, 100KB, 200KB) as input to the Web Inferencing workflow (deployed both on OL and OF); different document sizes indicate different sizes in text content. The text document is encapsulated in a HTTP request, this is sent to both OL and OF controllers to trigger the execution of the workflow starting from Text Summarizer. We discuss the execution times of functions, end-to-end latency, overheads of the platform in the results.

The plots in the Fig. 6.3 shows the end-to-end execution times for the web inferencing workflow for different input sizes in OF and OL. The X-axis indicates the input-size for the workflow, the Y-axis indicates the latency in milliseconds. The end-to-end times show an increase with increase in input size. The plots in Fig. 6.3 also shows the cumulative function execution times to be increasing with increase in input-size. The platform overheads show a similar trend in OF and

OL, and both are considerably high (OF higher than OL). The plots in Fig. 6.3 shows the breakup of platform overheads as inter-function overheads between TextSummarizer → ProfanityDetection, and ProfanityDetection → SpamDetection. The overheads remain similar both in OL and OF, for different values across the input, as the text document sizes are of similar range (50KB to 200KB).

(a) End-To-End Latency (b) Function Execution Times

(c) Function Execution Times Split (d) Inter-function Latency

Fig. 9. Web Inferencing Scenario

A Discussion

A.1 Intrinsic Overheads, Domain Specific Applications and Serverless Platforms

While the serverless paradigm provides ease-of-use for the users and automatic deployment of functions, The existing open-source platforms exhibit high platform overheads for execution each independent functions as observed in the experiments. Applications from the web-domain which have relaxed QoS metrics may tolerate these overheads, especially the ones which are not user-facing, such as spam detection, profanity detection. However, the network services designed for traffic from applications such as VoIP, Video calling require much lower overheads and hence lower processing times. Towards this effect, the serverless platforms need mechanisms to lower the platform overheads to meet the QoS requirements of such applications. Also, not all functions in an application can

be stateless and short-lived, few front-line services (for instance load-balancers) need to be persistent services. This brings the need for smart orchestration of persistent and temporary services in a serverless paradigm.

A.2 Domain-specific requirements, QoS metrics, and Serverless Platforms

Each application presents a unique set of demands whilst showing a good fitness for the serverless computing paradigm. However, the existing platforms are limited in their ability to consider domain-specific requirements or QoS metrics in their function deployment, resource management, and provisioning mechanisms. For instance, variations in input sizes in the Image analytics domain show a variability in execution times. However, in the existing open source platforms, all the function execution requests are invoked with peak-resource assignment (resource specification that is assigned to meet the QoS metric for the largest input size for the function). This can be cost-inefficient since all the inputs may not be the largest input. Also, certain data privacy regulations may require specific functions to execute at Edge worker nodes (on-premises Edge data center). To address such requirements, there is a need for QoS specification framework, to express application requirements, their QoS metrics which can drive resource provisioning techniques, and function deployment decisions. Certain applications from the Network domain such as VoIP and Video calls expect ordered delivery of packets as a semantic requirement, but the existing open source platforms do not guarantee ordered execution of requests [1]; this leads to a violation of the QoS requirement of such application.

A.3 Feature gaps

Based on experiments and analysis, we present some of the shortcomings in the existing open source FaaS platforms OF and OL in Table A.2. The resource specification in existing FaaS platforms are fixed at the time of function deployment, each invocation of a function executes with the same resource specification. It does not change with input size or any other function-assisted input, even though a function's performance is sensitive to such parameters (as in case of Image Resizing where execution time increases with increasing image/input size). Both the platforms exhibit high overheads, and forks a new process upon each request for execution prompting a cold start from the platform. OF and OL both does not support function sequences for applications, the users have to manually manage their workflows with an external data-store. The resource management does not take application's QoS metrics (latency, data privacy, cost, etc.,) into consideration in its provisioning decisions. Another issue is that in a resource pool comprising heterogeneous devices (Xeon servers of different generations, Raspberry Pi-Xeon Class devices), the resource specification does not take into account the compute capacity of worker nodes, this can cause variability in function execution times. Also, the existing platforms do not guarantee request ordering which maybe a QoS requirement for specific class of applications as

Features	**OpenFaaS**	**OpenLambda**
Resource Specification	Fixed, assigned for peak input	Fixed, assigned for peak input
Platform Overheads	High (order of 100s of ms)	High (order of 100s of ms)
Cold Starts	Forks a new process for each request	Forks a new process for each request
Application Workflows (Function Chaining)	No Support	No Support
Support for QoS metrics	QoS metrics are not considered in function deployment, resource management	QoS metrics are not considered in function deployment, resource management
Request Ordering	Function execution order is guaranteed	Function execution order is guaranteed
Support for Resource Heterogeneity	Does not take resource heterogeneity in provisioning decisions	Does not take resource heterogeneity in provisioning decisions
Request Ordering	Function execution order is guaranteed	Function execution order is guaranteed
Language support	Python, Docker, Go, Java, C#, NodeJS	Python, Docker

Table 2. Feature Gaps in OpenFaaS and OpenLambda

discussed in section A.2. Finally, both the platforms OF and OL support popular programming language runtime, and custom application runtime through Docker containers.

B Conclusion and Future Work

In this study, we surveyed three application domains Video surveillance, Web Inferencing, and NFV. The application characteristics, workloads, and their QoS requirements were presented; their fitness to serverless paradigm were also discussed. We deployed three representative workflows one from each domain, and few individual benchmarking functions on two widely used open source FaaS platforms OpenFaaS and OpenLambda. Our evaluation brings out the intrinsic platform overheads in existing Open Source platforms, rigidity in resource specification, absence of QoS aware resource specification (from users), function deployment and resource provisioning at the platform. As a part of future work, we plan to benchmark commercial Serverless providers such as AWS Lambda and Azure functions. Our study highlights the necessity to bring out QoS aware serverless platforms, which enables specifications of domain specific application requirements and use it drive resource provisioning and function deployment.

Appendix

C OpenLambda : Deployment steps

OpenLambda [9] is an open-source serverless platform for research prototyping, the overview of OpenLambda is shown in the Fig. 10. The workflow involves the user uploading the function to the "code store" or "function registry" which is a directory path in the worker node of OpenLambda (1), the template for execution is described in [3].

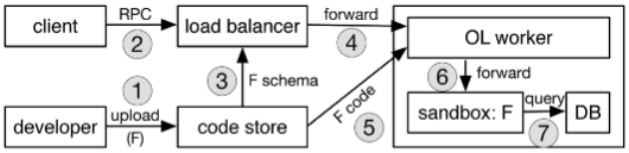


Fig. 10. OpenLambda architecture Overview

C.1 Workflow

Users can trigger function executions using simple HTTP requests, the payload is encapsulated in JSON (2), the request is intercepted by the Load balancer which retrieves (3) the details of the function (which function to execute), it identifies a worker and forwards the request (4), the worker fetches the function code from the registry (5) and forwards the request containing data to the function sandbox (6) which performs the execution and outputs the data to a data-store (7).

C.2 Support for function Workflow

The OpenLambda does not support function sequences/workflows out of the box. We provide support for composing Directed Acyclic Graph (DAG) workflows based on Apache OpenWhisk architecture [17]; OpenWhisk was chosen as reference since it provides support for workflows inherently in its architecture. CouchDB is chosen as the data-store as it supports storage and retrieval of native JSON, also it is conforms to the data store in the OpenWhisk architecture. After the completion of a function, the output is stored in the DB (data-store), which sends a complete request (event) to the Kafka Event manager which retrieves the input for the next function, and sends a new function execution request independently to the OL loadbalancer.

D OpenFaaS : Deployment steps

OpenFaaS provides a platform for deploying and managing individual functions, which are small, self-contained pieces of code that can be triggered by events. The following diagram gives the overview of the architecture of OpenFaaS.

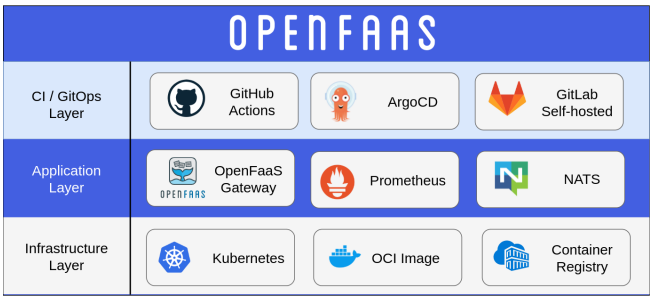


Fig. 11. OpenFaaS architecture Overview

- Portable functions platform - run functions on any cloud or on-premises.
- Write functions in any language and package them in Docker/OCI format containers.
- Easy to use - built-in UI, powerful CLI and one-click installation
- Scale as you go - handle spikes in traffic, and scale down when idle

Infrastructure Layer

- Functions operate within Pods, managed by containerd or Docker.
- Each function is stored as an immutable artifact in a container registry, deployable to the OpenFaaS gateway via REST API, UI, or CLI.
- Kubernetes allows function scaling across multiple hosts, while faasd offers a simpler alternative for smaller installations.

Application Layer

- The OpenFaaS gateway provides a REST API for function management, metrics recording, and scaling.
- Optional usage of NATS for asynchronous function execution and queuing.
- Prometheus handles metrics and facilitates auto-scaling.
  The combined components within OpenFaaS (Prometheus, Linux, OpenFaaS, NATS, and Kubernetes) are termed The PLONK Stack, capable of running event-driven functions and traditional HTTP-based microservices.

CI/GitOps Layer

OpenFaaS not only runs functions but also supports HTTP microservices. Each workload gets built into a container image and published in a container registry. During development, this is often managed manually using the faas-cli.

D.1 Workflow

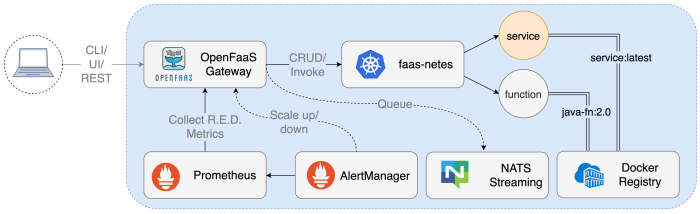


Fig. 12. OpenFaaS conceptual workflow

- The Gateway is accessible via its REST API, CLI, or UI. While default routes are exposed for all services/functions, custom domains can be utilized for individual endpoints.
- Prometheus collects and provides metrics through the Gateway's API, enabling auto-scaling.
- By altering the function URL from /function/NAME to /async-function/NAME, invocations can be queued using NATS Streaming.

D.2 Watchdog

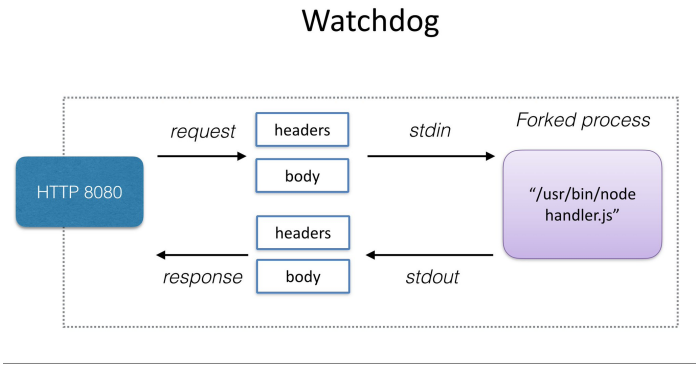


Fig. 13. Classic Watchdog

Within OpenFaaS, "classic watchdog" and "of-watchdog" refer to different approaches for managing serverless function execution and scaling. They ensure efficient and correct function execution. Let's explore both:

Classic Watchdog: The "classic watchdog" is an older method used in OpenFaaS for monitoring the execution of functions. Here's how it works:

- Initiates a child process for function execution upon invocation.
- Terminates function execution if a heartbeat isn't received within a specified timeout period.
- Manages function execution, monitors resource usage, and cleans up post-execution.

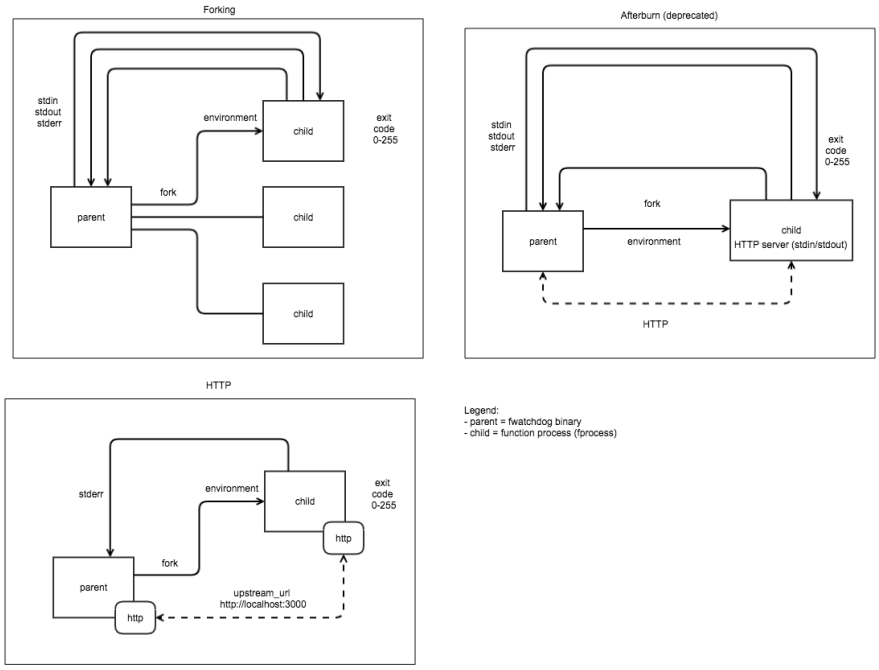


Fig. 14. of-watchdog

OF-Watchdog:

- The "of-watchdog" (or "the watchdog of watchtowers") is a more modern and efficient approach introduced to address the limitations of the classic watchdog. Here's how it works:
- Operates with a single "watchtower" per function namespace, managing all functions within it.
- Functions as an HTTP reverse proxy, receiving function invocation requests.

- Initializes function executions in separate containers using Docker or another container runtime.
- Efficiently manages container resources, monitors execution, and auto-scales based on activity.

Deploying a function - Prerequisites

- Docker, OpenFaaS, faas-cli

1. Write your function

Create a folder for your work:

```
$ mkdir -p ~/functions && \
cd ~/functions
```

Now let's create a new Python function using the CLI:

```
$ faas-cli new --lang python hello-python
```

This creates three files for you:

```
hello-python/handler.py
hello-python/requirements.txt
hello-python.yml
```

Let's edit the handler.py

```
import requests
import json

def handle(req):
result = {"found": False}
json_req = json.loads(req)
r = requests.get(json_req["url"])
if json_req["term"] in r.text:
result = {"found": True}

print json.dumps(result)
```

The 'handle' function serves as an endpoint capable of processing input (req) in both text and JSON formats. Once processed, the return statement sends the output back to the client using the OpenFaaS gateway.

Every function needs specification within a YAML file. This file guides the CLI regarding what to build and deploy onto the OpenFaaS cluster.

Checkout the YAML file hello-python.yml:

```
provider:
name: openfaas
gateway: http://127.0.0.1:8080

functions:
```

```
hello-python:
lang: python
handler: ./hello-python
image: hello-python
```

gateway - This allows us to define a remote gateway if necessary, specifying the programming language and the location of our handler within the filesystem.

functions - This section outlines the functions within our stack.

lang: python - Although Docker is used in the background to package the function, it's not mandatory to create a Dockerfile unless desired.

handler - This denotes the folder or path leading to the handler.py file and any additional source code required.

image - Here, we specify the Docker image name. If you intend to push to Docker Hub, modify the prefix from hello-python to include your Docker Hub account.

2. Deploy your function

We need to port-forward the OpenFaas gateway service to the localhost port

```
kubectl port-forward -n openfaas svc/gateway 8080:8080 &
```

Build

```
faas-cli build -f hello-python.yml
```

Push

```
faas-cli push -f hello-python.yml
```

Login

```
PASSWORD=$(kubectl get secret -n openfaas basic-auth -o jsonpath="
    {.data.basic-auth-password}" | base64 --decode; echo)
env | grep PASSWORD
echo -n $PASSWORD | faas-cli login --username admin --password-
    stdin
```

Open browser and navigate to http://127.0.0.1:8080. Copy your password and paste it in the prompted place.

Deploy

```
faas-cli deploy -f hello-python.yml -g http://127.0.0.1:8080
```

The function is expected to be deployed within the OpenFaaS interface. It may take some time for the status to indicate "Ready".

You can either invoke the function using the UI or through the cURL also:

```
$ curl 127.0.0.1:8080/function/hello-python --data-binary '{
"url": "https://github.com/ipanditi",
"term": "docker"
}'
```

Here's the result

```
{"found": true}
```

D.3 Additions for Function Chaining

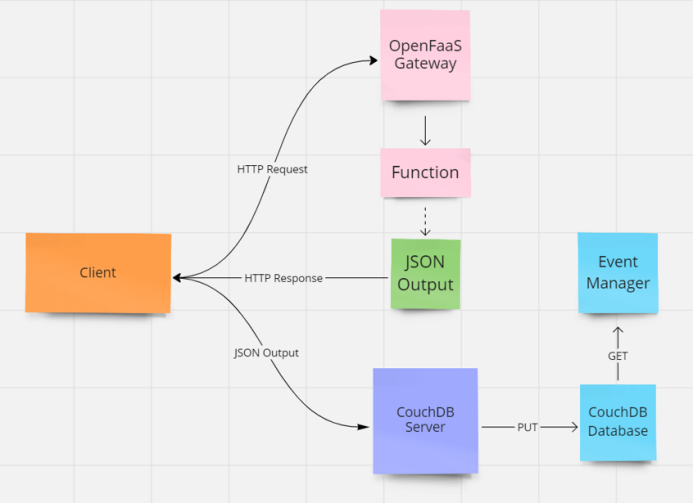


Fig. 15. Function chaining workflow

The sequence comprises several steps initiated by an HTTP request and executed using OpenFaaS and associated tools:

1. Request from Client

A client forwards an HTTP request to the OpenFaaS gateway, specifying the function name as http://127.0.0.1:8080/function/fname.

2. Processing by Gateway

The OpenFaaS gateway manages the received HTTP request, utilizing Watchdog, and invokes the designated application.

3. Handling the Response

The application generates a JSON output as a response. This response is then dispatched to a server and stored within a CouchDB database.

4. Management of Events

Apache Kafka, functioning as the event manager, retrieves the prior function name from CouchDB and initiates subsequent requests.

5. Sequential Execution

This cycle persists until all functions within the chain are executed, with each function triggering the next based on the event-driven architecture.

References

1. Openwhisk actions. https://github.com/apache/openwhisk/blob/master/docs/actions.md (2023)
2. Bruschi, R., Davoli, F., Lago, P., Pajo, J.F.: Model-based analytics for profiling workloads in virtual network functions. In: 2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). pp. 916–921 (2017)
3. Caraza-Harter, T.: Deploying an openlambda worker. https://github.com/open-lambda/open-lambda/blob/main/docs/worker/getting-started.md (2023)
4. Deri, L.: Open source voip traffic monitoring. In: Proceedings of SANE. vol. 2006 (2006)
5. Dinh-Xuan, L.: Quality of Experience Assessment of Cloud Applications and Performance Evaluation of VNF-Based QoE Monitoring. doctoralthesis, Universität Würzburg (2018)
6. Fonseca, H., Cruz, T., Simes, P., Monteiro, E., Silva, J., Gomes, P., Centeio, N.: A comparison of classification techniques for detection of voip traffic. In: 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies. pp. 117–122 (2014)
7. Gu, Y., Hu, Y., Ding, Y., Lu, J., Xie, J.: Elastic virtual network function orchestration policy based on workload prediction. IEEE Access 7, 96868–96878 (2019)
8. Gunleifsen, H., Kemmerich, T.: Security requirements for service function chaining isolation and encryption. In: 2017 IEEE 17th International Conference on Communication Technology (ICCT). pp. 1360–1365 (2017)
9. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with openlambda. In: 8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16) (2016)
10. Kim, J., Lee, K.: Functionbench: A suite of workloads for serverless cloud function service. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). pp. 502–504 (2019)
11. Laghrissi, A., Taleb, T., Bagaa, M., Flinck, H.: Towards edge slicing: Vnf placement algorithms for a dynamic amp; realistic edge cloud environment. In: GLOBECOM 2017 - 2017 IEEE Global Communications Conference. pp. 1–6 (2017)
12. Limna, T., Tandayya, P.: Video surveillance as a service cost estimation and pricing model. In: 2015 12th International Joint Conference on Computer Science and Software Engineering (JCSSE). pp. 174–179 (2015)
13. Neal, D.J., Rahman, S.: Video surveillance in the cloud? CoRR abs/1512.00070 (2015), http://arxiv.org/abs/1512.00070
14. Nevsky: Traffic surveillance. https://figshare.com/articles/dataset/Nevsky_prospect_traffic_surveillance_video_MOOL-cases_hours_/5841267/2?file=10387548, year = 2017
15. OFC: Openfog reference architecture for fog computing. "https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf" (2017)

16. OpenFaaS: https://docs.openfaas.com/reference/yaml/ (2023)
17. OpenWhisk, A.: Apache openwhisk. https://github.com/apache/openwhisk (2023)
18. de Saxc, H., Oprescu, I., Chen, Y.: Is http/2 really faster than http/1.1? In: 2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). pp. 293–299 (2015)
19. Singhvi, A., Banerjee, S., Harchol, Y., Akella, A., Peek, M., Rydin, P.: Granular computing and network intensive applications: Friends or foes? In: Proceedings of the 16th ACM Workshop on Hot Topics in Networks. p. 157163. HotNets-XVI, Association for Computing Machinery, New York, NY, USA (2017), https://doi.org/10.1145/3152434.3152450
20. Tian, Y., Babcock, R., Taylor, C., Ji, Y.: A new live video streaming approach based on amazon s3 pricing model. In: 2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC). pp. 321–328 (2018)